SYNTHESIS AND OPTIMIZATION OF INTERFACES BETWEEN HARDWARE MODULES WITH INCOMPATIBLE PROTOCOLS

Vassilis Androutsopoulos, TJW Clarke and DM Brookes

Department of Electrical and Electronic Engineering, Imperial College
London SW7 2BT, UK

ABSTRACT

In this paper, we present a new algorithm that performs automatic interface synthesis between two synchronous hardware modules with incompatible data communication protocols. We introduce the Data Path State Machine (DPSM) which captures data path dependencies. This allows control logic for data paths to be synthesized which is optimized for bandwidth over multiple transactions.

1. INTRODUCTION

The recent advent of Systems-On-Chip products, the growing complexity of designs and stringent time to market pressures are all factors for the so called *design productivity gap*. Unsurprisingly, pressures have therefore been exerted on EDA companies to develop tool environments to encourage the reuse of previous designs. The increasing reuse of RTL hardware blocks makes the interfacing of RTL hardware blocks important. Communication between these blocks is made possible if proper interface circuits are introduced. Manually adapting these interfaces is a tedious and error prone process. Instead, methods and algorithms to automatically synthesize interfaces need to be developed.

The problem can be expressed as: *given the producer and consumer data communication protocols and a description of the data path that interfaces the two sides, generate an optimal (in terms of performance) interface machine automatically that will synchronize and preserve the meaning of the data between the two sides.*

Passerone [PRSV98] showed that if protocols are represented naturally as Deterministic Finite Automata (DFA), the product FSM can be pruned to implement interface control logic. Passerone has argued that DFAs are not as easy for designers to use for protocol specification as regular expressions. In this paper we will assume that the protocol specifications can be translated automatically into equivalent DFAs as shown by [PRSV98]. We will address the problem of automatic interface synthesis from protocol DFAs.

Passerone's algorithm uses acyclic DFAs to simplify synthesis, and so does not optimize latency and bandwidth over multiple transactions. Furthermore it does not consider data path issues. We present a new algorithm that deals with more realistic interface synthesis in which protocols are represented by cyclic DFAs. The introduction of a Data Path State Machine (DPSM), capturing data path dependencies, allows control logic for data paths to be optimized for bandwidth.

The rest of the paper is organized as follows: Sect.2 gives a brief description of previous and related work, Sect.3 presents terminology required in the rest of the paper, Sect.4 presents the protocol specification and DPSM formalisms, Sect.5 presents the algorithm. Finally, Sect.6 describes the results and Sect.7 concludes the paper and presents direction for future work.

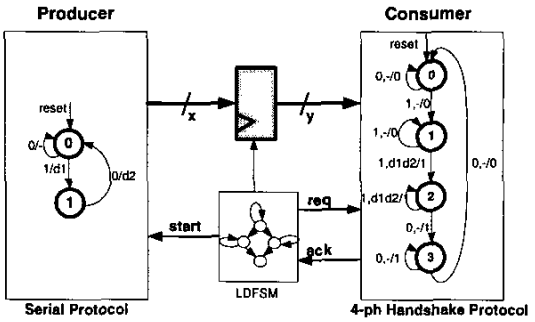

Figure 1: Problem Definition

2. RELATED WORK

Interface synthesis has been addressed in a broad range of literature. The STG is introduced in [Bor88] as a means to establish synchronization between synchronous and/or asynchronous components. However the protocol specifications are too low level (timing diagrams) and the correspondence between the different pieces of data items are not resolved automatically.

The authors in [AM91] describe the protocols using 2 verilog FSMs and a non-deterministic cartesian product is obtained which forms the interface. This is determinized by using a 3rd machine called the C-machine which describes the intended behavior of the interface. The method does not solve the data correspondence problem mentioned above and does not consider any data path issues.

Passerone et al. [PRSV98] describe the protocols using Regular expressions. These are translated into finite automata which are then synthesized into FSMs using a product algorithm which resolves the pseudo non-determinism that arises by making the composition causal, non-deadlocking and optimal in terms of its latency. It solves the data correspondence problem but is limited in form of communication i.e. only a single transaction, point-to-point communication and common clock are assumed.

More recently there have been efforts by [PCPK00] to generate hardware interfaces with both sides operating at different clock frequencies by inserting additional states and edges to the product

0-7803-7761-3/03/$17.00 ©2003 IEEE

FSM. The authors in [SG02] have recently proposed an interface architecture with 3 FSMDs (one for each of the producer, consumer and queue) and a data path consisting of a queue which they believe is general enough to accommodate any component protocols. The protocols are specified using FSMDs and the synthesis algorithm is responsible for mapping these onto the FSMDs on the target architecture. The algorithm does not address the data correspondence problem.

3. PRELIMINARY TERMINOLOGY

If $c_1, c_2, ..., c_n$ are the control ports associated with a certain protocol, assuming values from the sets $\sigma_1, \sigma_2, ..., \sigma_n$, the **control set** of the protocol is defined as the product $C = \prod_{i=1}^{n} \sigma_i$. Elements from the control set are called **control symbols**.

If $d_1, d_2, ..., d_q$ are the data ports associated with a certain protocol, assuming values from the sets $\rho_1, \rho_2, ..., \rho_q$, the **data set** of the protocol is defined as the product $D = \prod_{i=1}^{q} \rho_i$. Elements from the data set are called **data symbols**.

The **alphabet** Σ of the protocol is defined as the product $C \times D$. The elements of the alphabet are called symbols. A **formal language** over Σ is a set of strings of symbols from Σ. A **protocol** is a formal language over Σ. In other words a protocol is a set of strings of symbols from Σ where each string of symbols represents a legal manifestation of a certain transaction or behaviour.

Elements from the protocol set are called **tokens**. A token represents a complete communication between the producer and consumer. The set of data symbols associated with the token are known as the **data type**. In a bus transaction a token is broken down into a series of **sub-tokens**. Each sub-token consists of a string of data and control symbols. The data symbols are associated with the data bus and the control symbols are associated with the control signals.

Figure 2 illustrates the producer and consumer **Data Flow States** (DFS). Boolean expressions involving DFS form the conditions on the transitions of the Data Path State Machine.

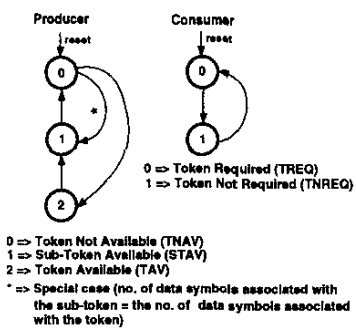

Figure 2: Producer and Consumer Data Flow States

4. INTERFACE SPECIFICATION

A DFA is translated into the equivalent Labelled pseudo Non Deterministic Finite State Machine by considering the behaviour of the input and output wires separately.

Definition 1 A Labelled pseudo Non Deterministic Finite State Machine (LNDFSM) is a directed graph defined by either of the following tuples depending on whether the hardware block conforming to the communication protocol is a producer or a consumer.

$$LNDFSM_p :=< S, I, O \cup V, L, \delta, \lambda, P, F, s_0 >$$

$$LNDFSM_c :=< S, I \cup V, O, L, \delta, \lambda, P, F, s_0 >$$

S denotes the set of protocol states with $s_0 \in S$ being the reset state. I represents the finite input control space, O represents the finite output control space and $V \subseteq D$ defines the set of storage variables. L is the set of transition labels $(\alpha/\beta, \gamma)$ (for a producer) or $(\alpha, \gamma/\beta)$ (for a consumer) where $\alpha \in I$, $\beta \in O$, γ is a label distinguishing between the different storage variables in the transaction. $\delta : S \times I \rightarrow S$ is the next state function and λ is the output relation denoted by the characteristic function $\lambda : I \times S \times O \rightarrow B$ where $B = \{0, 1\}$. $F \subseteq S$ denotes the set of final states. A final state is the state which represents the entire data type being sent/received for the last time in the transaction. For a linear FSM $\mid F \mid = 1$. For a non-linear (branching) protocol $\mid F \mid \geq 1$. P is the set of transaction parameters (i.e no. of data symbols associated with the sub-token and token).

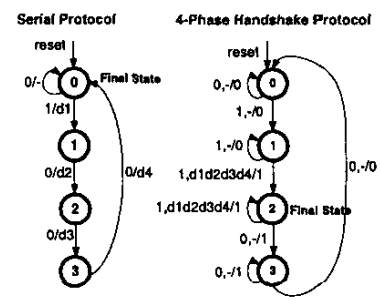

Figure 3: LNDFSMs for Producer and Consumer protocols

Example linear LNDFSMs for a non-stallable serial protocol (acting as the producer) and a 4-phase handshake protocol (acting as the consumer) are shown in figure 3. The serial protocol initially waits for the environment to set its input control signal to 1. An associated data symbol $d1$ is waiting to be placed onto the data bus. Once a 1 is received from the environment, the producer puts the data onto the bus. The control signal then goes to 0 one clock cycle later and another data symbol $d2$ is placed onto the bus. The last data symbol $d4$ in the data type, is placed onto the data bus two clock cycles later.

The 4-phase handshake protocol initially waits for a request signal from the environment. After the environment sets the request signal, it waits for the hardware block to assign the acknowledge signal. After the acknowledge signal is received, the environment puts the data onto the bus. The hardware block reads the data from the bus until the environment drops the request signal.

Definition 2 A Data Path State Machine (DPSM) is a directed acyclic graph defined by the tuple

$$DPSM :=< R, \delta, C, r_0, F >$$

where R denotes the set of protocol states, with $r_0 \in S$ being the initial state and $F \in S$ the final state, $\delta \subseteq S \times S$ is the set of state transitions, and C is the set of all possible DFS conditions for the producer and the consumer. The states in the DPSM emphasize

the acceptance or rejection of a sequence of DFS conditions. The edges are labelled with these conditions.

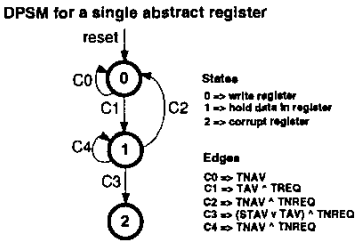

Figure 4: DPSM for a single abstract register

The DPSM for a single abstract register wide enough to store the entire data type is illustrated in Figure 4. We assume that the register is tied to the producer and that the various data is clocked into the register when it is first made available. State 1 in the DPSM is entered upon when the producer has written the entire data type to the register and is waiting for the consumer to relinquish the use of the data contents in the register. If at anytime during which the data path is in state 1, the producer writes a new set of data symbols to the register, the contents of the register will be corrupted and the consumer will thereafter read incorrect data.

A possible refined version of the abstract register is shown in Figure 5. It will consist of D modified registers where D represents the size of the data type. These registers are modified to optimize latency. X and Y represent the input and output port sizes respectively in terms of the number of data symbols that can be associated with them.

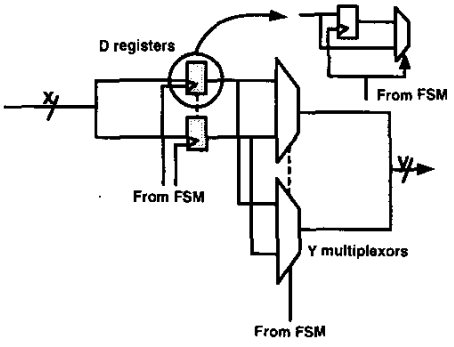

Figure 5: Data path architecture

5. SYNTHESIS ALGORITHM

The inputs to the synthesis algorithm are the LNDFSMs for the producer and consumer data communication protocols and the DPSM for the abstract register. The resulting composition is the Labelled Deterministic Finite State Machine (LDFSM), a subset of the cartesian product of the 2 LNDFSMs and DPSM. The LDFSM will control the data flow between the producer and consumer, and preserve the meaning of the data according to their LNDFSM specifications. The LDFSM coupled with the abstract register of figure 5 will form the interface.

Definition 3 A Labelled Deterministic Finite State Machine (LDFSM) is defined by the tuple

$$LDFSM := < Q, I \cup R \cup V, O \cup V, \Lambda, \Delta, \lambda, q_0, F >$$

where $Q \subseteq S_p \times S_c \times R$ denotes the set of states with $q_0 \in S$ being the reset state and $F \subseteq Q$ the set of final states. I represents the finite input space, O the finite output space, R represents the set of possible data path states, $V \subseteq D$ defines the set of storage variables, $\Lambda \subseteq L_p \times L_c$ is the set of transition labels $(\alpha, \beta/\gamma, \delta)$ where α and γ are the input and output control symbols and β and δ are the input and output data labels distinguishing between the different data symbols. $\Delta \subseteq Q \times Q$ is the set of state transitions, $\lambda : I \cup R \times S \rightarrow O$ is the output relation.

The LDFSM is computed by using a modified version of Passerone's product computation algorithm [PRSV98] (see Figure 6). The algorithm has been re-implemented in approximately 2000 lines of C-code and modified to handle multiple transactions. The algorithm accepts the DPSM and the producer and consumer data communication protocols as input. The DPSM is used by the algorithm to capture the desired data dependencies for the given data path. In particular what is new in our implementation of the algorithm is the possibility for the user to specify any arbitrary data path behaviour in terms of the DPSM. It returns the LDFSM composition if one exists, which is optimal in terms of the cycle length.

Two auxiliary data structures are created to assist the product machine computation: Stack and Fail Pool. The stack marks the explored path and is used to detect loops and to avoid endless computation. The fail pool is used to record failed states.

The Explore function returns one of three objects: Success, Fail, Loop. The explore function returns Success for a transition that will definitely lead to a successful transfer of data and Loop if the state already exists on the stack and has already been visited. Fail is returned if the interface is in a state where the data transfer is non-causal, the buffer has overflown or is uncontrollable (i.e. will lead to either one of these states in an unfriendly environment).

The product is computed by performing a depth first recursive search with backtracking on all possible states in the product machine. The required subset of the product machine is constructed by starting from no states and then adding states. Each new state in the tree is explored and the pseudo non-determinism that arises is resolved by choosing the transitions which make the resulting composition causal, controllable and optimal in terms of its cycle length. In particular, we define a final state as a state which contains backedges to states previously marked on the stack and the minimum cycle length of all the states are computed with reference to these final states. If a final state contains multiple backedges which are non-deterministic, the backedge which results in the smaller cycle length is chosen.

6. EXPERIMENTAL RESULTS AND DISCUSSION

In the first experiment, a non-stallable serial protocol is interfaced to a 4-phase request acknowledge protocol (see figure 3). In the second experiment, the same protocols are interfaced to one another as in the first experiment only this time the order in which the producer sends the data symbols to the consumer is reversed. In the third experiment, the 4-phase request acknowledge protocol is

```
explore (current state) {
  if (state is on Stack)
    return loop
  if (overflow v non-causal)
    return fail
  if (state is on Fail Pool)
    return fail
/* Begin new exploration */
Push state on stack
  for all pairs of transitions {
    evaluate causality
    compute new DPSM state
    compute new state
    explore (new state)
    save exploration result
    if (exploration result is Loop){
      redirect edge to point to ancestor on stack
    }
    else if (exploration result is Success)
      update current state outedge with new state
    }
    Pop state from stack
    Determine minimum cycle length from current state
    Choose among non-deterministic transitions
    Compute Exploration result
    Update Fail Pool
    return exploration result
  }
```

Figure 6: Explore Function Pseudo Code

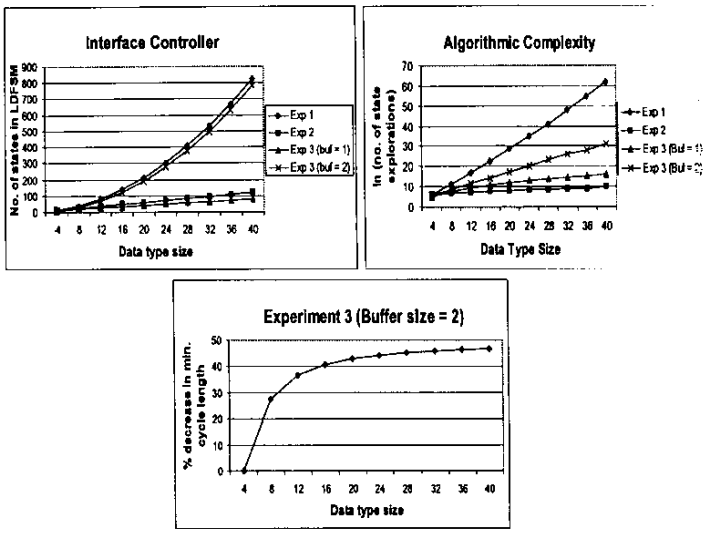

Figure 7: Experimental Results

interfaced to the non-stallable serial protocol. This involves translating the 4-phase request acknowledge protocol input signal *ack* (as observed by the interface) into an output, and the output signal *req* into an input. The non-stallable serial protocol transfers one data symbol at a time without interruption until the entire token has been transferred. The start of the next transaction is regulated by the interface. The request acknowledge side uses a bus four time larger in size which is regulated by the request-acknowledge signals.

As expected, the resulting controller FSMs are cyclic. For the same examples, Passerone's algorithm [PRSV98] produced acyclic FSMs because only a single transaction is considered. The main results are summarized in figure 7. The first experiment contains many states because the output of the data is concurrent to the input of the data. The possibility to stop and resume the protocol anywhere during the data transfer gives rise to a large number of states. Also note that concurrency leads to an exponential increase in the number of state explorations with increasing data type size. This problem can be overcome and is currently being dealt with. Clearly, one way to significantly reduce the number of explorations is to record the explored successful product states along with their minimum cycle length whilst performing the recursive search. In this way re-exploration of the same states can be avoided. Despite the current large no. of explorations, the generated controller is optimal in terms of bandwidth.

In the second experiment, the interface will have to wait to the end of the input phase to begin the output phase. Less choice leads to fewer states than in the first experiment. Also note that there are far fewer state explorations. This is because all the illegal states are found to be close to the initial state. The existence of a protocol violation in the third experiment reduces the number of choices for the interface machine which results in far less states than the first machine. The resulting interface is non-optimal in terms of the bandwidth. To optimize the bandwidth, the capacity of the interconnecting buffer was increased to store two tokens. The number of states increases with increasing buffer size. This is because increasing the buffer size increases the state space and also reduces the prospect for a protocol violation.

7. CONCLUSIONS AND FUTURE WORK

We presented a novel extension to Passerone's algorithm [PRSV98] which supports multiple transactions. The algorithm finds the optimal solution in terms of the bandwidth. Still, there are a number of interesting extensions. An obvious extension to this work is to devise a DPSM formalism to synthesize optimal control logic for different register configurations, to allow more complex data path synthesis. Another interesting extension would be to extend the approach to optimize the interface for data transfer latency and to determine a means to generate the best interface in terms of both data transfer latency and bandwidth.

8. REFERENCES

[AM91] Janaki Akella and Kenneth McMillan. Synthesizing Converters between Finite State Protocols. In *Proc. of the International Conference on Computer Design (ICCD'91)*, pages 410 – 413, 1991.

[Bor88] Gaetano Borriello. A New Specification Methodology and its Application to Transducer Synthesis. Phd Thesis UCB/CSD 88/430, University of California, Berkeley, 1988.

[PCPK00] Bong-Il Park, Hoon Choi, In-Cheol Park, and Chong-Min Kyung. Synthesis and Optimization of Interface Hardware between IP's Operating at Different Clock Frequencies. In *Proc. International Conference on Computer Design (ICCD'00)*, pages 519 – 524, 2000.

[PRSV98] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic Synthesis of interfaces between incompatible protocols. In *Proceedings of the Design Automation Conference (DAC'98)*, pages 8 – 13, 1998.

[SG02] Dongwan Shin and Daniel Gajski. Interface Synthesis from Protocol Specification. Technical Report CECS-02-13, University of California, Irvine, April 2002.